AMD Core Math Library (ACML)

Version 2.6.0

Short Contents

Table of Contents

1 Introduction

The AMD Core Math Library (ACML) is a set of numerical routines tuned specifically for AMD64 platform processors (including Opteron™ and Athlon™64). The routines, which are available via both FORTRAN 77 and C interfaces, include:

- BLAS - Basic Linear Algebra Subprograms (including Sparse Level 1 BLAS);
- LAPACK - A comprehensive package of higher level linear algebra routines;
- FFT - a set of Fast Fourier Transform routines for real and complex data.

The BLAS and LAPACK routines provide a portable and standard set of interfaces for common numerical linear algebra operations that allow code containing calls to these routines to be readily ported across platforms. Full documentation for the BLAS and LAPACK are available online. This manual will, therefore, be restricted to providing brief descriptions of the BLAS and LAPACK and providing links to their documentation and other materials (see Chapter 3 [The BLAS], page 12 and see Chapter 4 [LAPACK], page 13).

The FFT is an implementation of the Discrete Fourier Transform (DFT) that makes use of symmetries in the definition to reduce the number of operations required from O(n*n) to O(n*log n) when the sequence length, n, is the product of small prime factors; in particular, when n is a power of 2. Despite the popularity and widespread use of FFT algorithms, the definition of the DFT is not sufficiently precise to prescribe either the forward and backward directions (these are sometimes interchanged), or the scaling factor associated with the forward and backward transforms (the combined forward and backward transforms may only reproduce the original sequence by following a prescribed scaling).

Currently, there is no agreed standard API for FFT routines. Hardware vendors usually provide a set of high performance FFTs optimized for their systems: no two vendors employ the same interfaces for their FFT routines. The ACML provides a set of FFT routines, optimized for AMD64 processors, using an ACML-specific set of interfaces. The functionality, interfaces and use of the ACML FFT routines are described below (see Chapter 5 [Fast Fourier Transforms], page 17).

Chapter 2 [General Information], page 2 provides details on:

- how to link a user program to the ACML;
- FORTRAN and C interfaces to ACML routines;
- how to obtain the ACML version and build information;
- how to access the ACML documentation.

A supplementary library of fast math and fast vector math functions (ACML_MV) is also provided with some 64-bit versions of ACML. Some of the functions included in ACML_MV are not callable from high-level languages, but must be called via assembly language; the documentation of ACML_MV (see Chapter 6 [Fast Vector Math Library], page 58) gives details for each individual routine.

2 General Information

2.1 Determining the best ACML version for your system

ACML comes in versions for 64-bit and 32-bit processors, running both Linux and Microsoft Windows®operating systems. To use the following tables, you will need to know answers to these questions:

- Are you running a 64-bit operating system (on AMD64 hardware such as Opteron or Athlon64)? Or are you running a 32-bit operating system?
- Is the operating system Linux or Microsoft Windows?
- Do you have the GNU compilers (g77/gcc) or compatible compilers (compilers that are interoperable with the GNU compilers) installed?
- Do you have the PGI compilers (pgf77/pgf90/pgcc) installed?
- Do you have the PathScale compilers (pathf90/pathcc) installed?
- On a 32-bit Windows machine, do you have Microsoft C, or Compaq Visual Fortran, or compatible compilers installed?
- Do you have a single processor system or a multiprocessor (SMP) system? The single processor version of ACML can be run on an SMP machine and vice versa, but (if you have the right compilers) it is more efficient to run the version appropriate to the machine.
- If you're on a 32-bit machine, does it support Streaming SIMD Extension instructions (SSE or SSE2)?

The ACML installation includes a binary utility that can help you find an answer to the last question. The utility lies in directory util, and is named cpuid.exe. It interrogates the processor to determine whether SSE and SSE2 instructions exist.

```
util/cpuid.exe
```

Under a Linux operating system, another way of finding out the answer to the last question is to look at the special file /proc/cpuinfo, and see what appears under the "flags" label. Try this command:

```
cat /proc/cpuinfo | grep flags
```

If the list of flags includes the flag "sse" then your machine supports SSE instructions. If it also includes "sse2" then your machine supports SSE2 instructions. If your machine supports these instructions, it is better to use a version of ACML which was built to take advantage of them, for reasons of good performance.

The method of examining /proc/cpuinfo can also be used under Microsoft Windows if you have the Cygwin UNIX-like tools installed (see http://www.cygwin.com/) and run a bash shell. Note that AMD64 machines always support both SSE and SSE2 instructions, under both Linux and Windows. Older (32-bit) AMD chips may support SSE but not SSE2, or neither SSE nor SSE2 instructions. Other manufacturers' hardware may or may not support SSE or SSE2.

If you cannot determine whether or not your machine handles SSE or SSE2 instructions, you may prefer to assume that it does not. If you link to a version of ACML that was built

to use SSE or SSE2 instructions, and your machine does not in fact support them, it is likely that your program will halt due to encountering an "illegal instruction" - you may or may not be notified of this by the operating system.

Once you have answered the questions above, use these tables to decide which version of ACML to link against.

Linux 64-bit

Number of processors	*Compilers*	*ACML install directory*
Single processor	GNU g77/gcc or compatible	`acml2.6.0/gnu64`
"	PGI pgf77/pgf90/pgcc	`acml2.6.0/pgi64`
"	PathScale pathf90/pathcc	`acml2.6.0/pathscale64`
Multi processor	PGI pgf77/pgf90/pgcc	`acml2.6.0/pgi64_mp`

Linux 32-bit

Number of processors	*Compilers*	*SSE supported*	*ACML install directory*
Single	GNU g77/gcc or compatible	SSE and SSE2	`acml2.6.0/gnu32`
"	"	SSE but no SSE2	`acml2.6.0/gnu32_nosse2`
"	"	No SSE or SSE2	`acml2.6.0/gnu32_nosse`
"	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.6.0/pgi32`
"	"	SSE but no SSE2	`acml2.6.0/pgi32_nosse2`
"	"	No SSE or SSE2	`acml2.6.0/pgi32_nosse`
Multiple	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.6.0/pgi32_mp`

Microsoft Windows 64-bit

Number of processors	*Compilers*	*ACML install directory*
Single processor	PGI pgf77/pgf90/pgcc	`acml2.6.0/win64`

Microsoft Windows 32-bit

Number of processors	*Compilers*	*SSE supported*	*ACML install directory*
Single processor	GNU g77/gcc	SSE and SSE2	`acml2.6.0/gnu32`
”	”	SSE but no SSE2	`acml2.6.0/gnu32_nosse2`
”	”	No SSE or SSE2	`acml2.6.0/gnu32_nosse`
”	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.6.0/pgi32`
”	”	SSE but no SSE2	`acml2.6.0/pgi32_nosse2`
”	”	No SSE or SSE2	`acml2.6.0/pgi32_nosse`
”	CVF/Microsoft C	SSE and SSE2	`acml2.6.0/win32`
Multi processor	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.6.0/pgi32_mp`

2.2 Accessing the Library (Linux)

2.2.1 Accessing the Library under Linux using GNU g77/gcc

If the Linux 64-bit g77 version of ACML was installed in the default directory, /opt/acml2.6.0/gnu64, then the command:

```
g77 -m64 driver.f -L/opt/acml2.6.0/gnu64 -lacml
```

can be used to compile the program driver.f and link it to the ACML.

The ACML Library is supplied in both static and shareable versions, libacml.a and libacml.so, respectively. By default, the commands given above will link to the shareable version of the library, libacml.so, if that exists in the directory specified. Linking with the static library can be forced either by using the compiler flag `-static`, e.g.

```
g77 -m64 driver.f -L/opt/acml2.6.0/gnu64 -static -lacml
```

or by inserting the name of the static library explicitly in the command line, e.g.

```
g77 -m64 driver.f /opt/acml2.6.0/gnu64/libacml.a
```

Notice that if the application program has been linked to the shareable ACML Library, then before running the program, the environment variable `LD_LIBRARY_PATH` must be set, for example, by the C-shell command:

```
setenv LD_LIBRARY_PATH /opt/acml2.6.0/gnu64
```

where it is assumed that libacml.so was installed in the directory /opt/acml2.6.0/gnu64 (see the man page for ld(1) for more information about `LD_LIBRARY_PATH`.).

The command

```
g77 -m32 driver.f -L/opt/acml2.6.0/gnu32 -lacml
```

will compile and link a 32-bit program with a 32-bit ACML.

The command

```
gcc -m64 -I/opt/acml2.6.0/include driver.c
    -L/opt/acml2.6.0/gnu64 -lacml -lg2c
```

will compile and link a 64-bit C program with a 64-bit ACML, using the switch "-I/opt/acml2.6.0/include" to tell the compiler to search directory /opt/acml2.6.0/include for the ACML C header file acml.h, which should be included by driver.c. Note that it is necessary to add the compiler run-time library -lg2c when linking the program.

2.2.2 Accessing the Library under Linux using PGI compilers pgf77/pgf90/pgcc

Similar commands apply for the PGI versions of ACML. For example,

```
pgf77 -tp=k8-64 -Mcache_align driver.f -L/opt/acml2.6.0/pgi64 -lacml
pgf77 -tp=k8-32 -Mcache_align driver.f -L/opt/acml2.6.0/pgi32 -lacml
```

will compile driver.f and link it to the ACML using 64-bit and 32-bit versions respectively. In the example above we are linking with the single-processor PGI version of ACML.

If you have an SMP machine and want to take best advantage of it, link against the PGI OpenMP version of ACML like this:

```
pgf77 -tp=k8-64 -mp -Mcache_align driver.f
      -L/opt/acml2.6.0/pgi64_mp -lacml
pgf77 -tp=k8-32 -mp -Mcache_align driver.f
      -L/opt/acml2.6.0/pgi32_mp -lacml
```

Note that the location of the ACML is now specified as pgi64_mp or pgi32_mp. The -mp flag is important - it tells pgf77 to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time. The -Mcache_align flag is also important - it tells the compiler to align objects on cache-line boundaries.

The commands

```
pgcc -c -tp=k8-64 -mp -Mcache_align -I/opt/acml2.6.0/include driver.c
pgcc -tp=k8-64 -mp -Mcache_align driver.o
      -L/opt/acml2.6.0/pgi64_mp -lacml -lpgftnrtl -lm
```

will compile driver.c and link it to the 64-bit ACML. Again, the -mp flag is important if you are linking to the PGI OpenMP version of ACML. The switch "-I/opt/acml2.6.0/include" tells the C compiler to search directory /opt/acml2.6.0/include for the ACML C header file acml.h, which should be included by driver.c. Note that in the example we add the libraries -lpgftnrtl and -lm to the link command, so that required PGI compiler run-time libraries are found.

2.2.3 Accessing the Library under Linux using PathScale compilers pathf90/pathcc

Similar commands apply for the PathScale versions of ACML. For example,

```
pathf90 driver.f -L/opt/acml2.6.0/pathscale64 -lacml
```

will compile driver.f and link it to the ACML using the 64-bit version.

The commands

```
pathcc -c -I/opt/acml2.6.0/include driver.c
pathcc driver.o -L/opt/acml2.6.0/pathscale64 -lacml -lpathfortran
```

will compile driver.c and link it to the 64-bit ACML. The switch "-I/opt/acml2.6.0/include" tells the C compiler to search directory /opt/acml2.6.0/include for the ACML C header file acml.h, which should be included by driver.c. Note that in the example we add the library -lpathfortran to the link command, so that the required PathScale compiler run-time library is found.

The 32-bit ACML libraries come in several versions, applicable to hardware with or without SSE instructions (Streaming SIMD Extensions), and it is important to link to a library that is appropriate to your hardware. The variant library versions are distinguished by being in directories with different names.

2.2.4 Accessing the Library under Linux using compilers other than GNU, PGI or PathScale

It may be possible to link to the g77/gcc versions of ACML using other compilers, if they are compatible with g77/gcc. An important thing to note is that you will need to link in required compiler run time libraries. An example using the 32-bit Intel FORTRAN compiler *ifc* might look like this:

```
ifc driver.f -L/opt/acml2.6.0/gnu32 -lacml /usr/lib/libg2c.so
```

where /usr/lib/libg2c.so is required to resolve g77 compiler run-time library symbols.

2.3 Accessing the Library (Microsoft Windows)

2.3.1 Accessing the Library under 32-bit Windows using GNU g77/gcc

Under Microsoft Windows®, for the g77/gcc version of ACML it is assumed that you have the Cygwin UNIX-like tools installed (see http://www.cygwin.com/), including the g77/gcc compiler and associated tools. Assuming you have installed the ACML in the default place, then in a DOS command prompt window, the command

```
g77 driver.f "c:\Program Files\AMD\acml2.6.0\gnu32\libacml.a"
```

can be used to link the application program driver.f to the static library version of the ACML.

The g77 version of the ACML Library is supplied in both static and shareable versions, libacml.a and libacml.dll, respectively. The command given above links to the static version of the library, libacml.a. To link to the DLL version, the command

```
g77 driver.f "c:\Program Files\AMD\acml2.6.0\gnu32\libacml.dll"
```

can be used. Notice that if the application program has been linked to the DLL version of the ACML Library, then before running the program, the environment variable `PATH` must have been set to include the location of the DLL, for example by the DOS command:

```
PATH="c:\Program Files\AMD\acml2.6.0\gnu32";%PATH%
```

where it was assumed that libacml.dll was installed in the directory "c:\Program Files\AMD\acml2.6.0\gnu32". Alternatively, the PATH environment variable may be set in the system category of the Windows control panel.

The command

```
gcc "-Ic:\Program Files\AMD\acml2.6.0\include" driver.c
    "c:\Program Files\AMD\acml2.6.0\gnu32\libacml.a" -lg2c
```

will compile driver.c and link it to the 32-bit g77/gcc version of ACML. The switch "-Ic:\Program Files\AMD\acml2.6.0\include" tells the gcc compiler to search directory "c:\Program Files\AMD\acml2.6.0\include" for the ACML C header file acml.h, which should be included by driver.c. Note that it is necessary to add the compiler run-time library -lg2c when linking the program.

2.3.2 Accessing the Library under 32-bit Windows using PGI compilers pgf77/pgf90/pgcc

To use the 32-bit Windows PGI version of ACML, use a command like

```
pgf77 -Mcache_align driver.f
      "c:\Program Files\AMD\acml2.6.0\pgi32\libacml.a"
```

or

```
pgcc -c "-Ic:\Program Files\AMD\acml2.6.0\include"
     -Mcache_align driver.c
pgcc -Mcache_align driver.o
      "c:\Program Files\AMD\acml2.6.0\pgi32\libacml.a"
      -lpgftnrtl -lpgsse1 -lpgsse2 -lm
```

Note that in the example we link the program with -lpgftnrtl -lpgsse1 -lpgsse2 -lm so that required PGI run-time libraries are located.

If you have an SMP machine and want to take best advantage of it, link against the PGI OpenMP version of ACML like this:

```
pgf77 -mp -Mcache_align driver.f
      "c:\Program Files\AMD\acml2.6.0\pgi32\libacml.a"
```

Note that the location of the ACML is now specified as pgi32_mp. The -mp flag is important - it tells pgf77 to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time. The -Mcache_align flag is also important - it tells the compiler to align objects on cache-line boundaries. The -mp flag is also required if you compile and link a C program to the pgi32_mp libraries.

2.3.3 Accessing the Library under 32-bit Windows using Microsoft C or Visual Fortran

To use the 32-bit Windows MSC/CVF version of ACML, use a command like

```
cvf /threads /libs:dll driver.f
      "c:\Program Files\AMD\acml2.6.0\win32\libacml_dll.lib"
```

where libacml_dll.lib is the import library for the ACML DLL,

or

```
cl "-Ic:\Program Files\AMD\acml2.6.0\include"
     /Gz /MD driver.c
     "c:\Program Files\AMD\acml2.6.0\win32\libacml_dll.lib"
```

where cvf is the Compaq Visual Fortran command line compiler and cl is the Microsoft C command line compiler. The flag /Gz used on the C compiler command line is important - it tells it to use the _stdcall calling convention rather than the default Microsoft C __cdecl calling convention. All ACML user-callable routines were built using _stdcall to ensure that the ACML DLL is easily accessible from any language compatible with that convention (for example, Microsoft Visual Basic or Microsoft C#).

ACML can also be linked from inside a development environment such as Microsoft Visual Studio or Visual Studio.NET. Again, it is important to get compilation options correct. The directory acml2.6.0\win32\examples\Projects contains a few sample Visual Studio project directories showing how this can be done.

Note that in both examples above we linked to a DLL version of ACML, and so before running the resulting programs the environment variable `PATH` must be set to include the location of the DLL, for example by the DOS command:

```
PATH="c:\Program Files\AMD\acml2.6.0\win32";%PATH%
```

where it is assumed that libacml_dll.dll was installed in the directory "c:\Program Files\AMD\acml2.6.0\win32". Alternatively, the PATH environment variable may be set in the system category of the Windows control panel.

ACML also comes as a static (non-DLL) library, named libacml.lib, in the same directory as the DLL. If you link to the static library instead of the DLL import library then there is no need to set the PATH.

2.3.4 Accessing the Library under 64-bit Windows

Under 64-bit versions of Windows, ACML 2.6.0 comes only as a static (.LIB) library for single processor machines (the library can be used on an SMP machine but will not take advantage of more than one processor). The compiler can be used with the PGI compilers pgf77/pgf90/pgcc or with the Microsoft C compiler, though with the latter compiler it is necessary also to link with PGI run-time libraries.

To link with the 64-bit Windows version of ACML, in a DOS command prompt use a command like

```
pgf77 driver.f c:/Program Files/AMD/acml2.6.0/win64/libacml.lib
```

or, for a C program,

```
pgcc driver.c -Ic:/Program Files/AMD/acml2.6.0/include
     c:/Program Files/AMD/acml2.6.0/win64/libacml.lib
     -lpgftnrtl -lm
```

Note that in the C example we link the program with -lpgftnrtl -lm so that required PGI run-time libraries are located.

To use the Microsoft C command line compiler, *cl*, use a command like this:

```
cl driver.c -Ic:/Program Files/AMD/acml2.6.0/include
     c:/Program Files/AMD/acml2.6.0/win64/libacml.lib
     c:/usr/pgi/win64/1.0/lib/libpgftnrtl.lib
     c:/usr/pgi/win64/1.0/lib/libpgc.lib
```

The references to libpgftnrtl.lib and libpgc.lib must point at the location of an installed copy of the PGI compilers.

2.4 ACML FORTRAN and C interfaces

All routines in ACML come with both FORTRAN and C interfaces. The FORTRAN interfaces typically follow the relevant standard (e.g. LAPACK, BLAS). Here we document how a C programmer should call ACML routines.

In C code that uses ACML routines, be sure to include the header file <acml.h>, which contains function prototypes for all ACML C interfaces. The header file also contains C prototypes for FORTRAN interfaces, thus the C programmer could call the FORTRAN interfaces from C, though there is little reason to do so.

C interfaces to ACML routines differ from FORTRAN interfaces in the following major respects:

- The FORTRAN interface names are appended by an underscore (except for the Windows 32-bit Microsoft C/Compaq Visual Fortran (CVF) version of ACML, where FORTRAN interface names are distinguished from C by being upper case rather than lower case - this is the default for the CVF compiler)
- The C interfaces contain no workspace arguments; all workspace memory is allocated internally.
- Scalar input arguments are passed by value in C interfaces. FORTRAN interfaces pass all arguments (except for character string *length* arguments that are normally hidden from FORTRAN programmers) by reference.
- Most arguments that are passed as character string pointers to FORTRAN interfaces are passed by value as single characters to C interfaces. The character string *length* arguments of FORTRAN interfaces are not required in the C interfaces.
- Unlike FORTRAN, C has no native *complex* data type. ACML C routines which operate on complex data use the types *complex* and *doublecomplex* defined in <acml.h> for single and double precision computations respectively. Some of the programs in the ACML examples directory (see Section 2.7 [Examples], page 11) make use of these types.

It is important to note that in both the FORTRAN and C interfaces, 2-dimensional arrays are assumed to be stored in column-major order. e.g. the matrix

$$A = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

would be stored in memory as 1.0, 3.0, 2.0, 4.0. This storage order corresponds to a FORTRAN-style 2-D array declaration A(2,2), but not to an array declared as a[2][2] in C which would be stored in row-major order as 1.0, 2.0, 3.0, 4.0.

As an example, compare the FORTRAN and C interfaces of LAPACK routine dsytrf as implemented in ACML.

FORTRAN:

```
void dsytrf_(char *uplo, int *n, double *a, int *lda, int *ipiv,
             double *work, int *lwork, int *info, int uplo_len);
```

C:

```
void dsytrf(char uplo, int n, double *a, int lda, int *ipiv,
            int *info);
```

C code calling both the above variants might look like this:

```
double *a;
int *ipiv;
double *work;
int n, lda, lwork, info;

/* Assume that all arrays and variables are allocated and
   initialized as required by dsytrf. */

/* Call the FORTRAN version of dsytrf. The first argument
   is a character string, and the last argument is the
   length of that string. The input scalar arguments n, lda
   and lwork, as well as the output scalar argument info,
   are all passed by reference. */
dsytrf_("Upper", &n, a, &lda, ipiv, work, &lwork, &info, 5);

/* Call the C version of dsytrf. The first argument is a
   character, workspace is not required, and input scalar
   arguments n and lda are passed by value. Output scalar
   argument info is passed by reference. */
dsytrf('U', n, a, lda, ipiv, &info);
```

2.5 Library Version and Build Information

This document is applicable to version 2.6.0 of ACML. The utility routine `acmlversion` can be called to obtain the major, minor and patch version numbers of the installed ACML. This routine returns three integers; the major, minor and patch version numbers, respectively.

The utility routine `acmlinfo` can be called to obtain information on the compiler used to build ACML, the version of the compiler, and the options used for building the Library. This subroutine takes no arguments and prints the information to the current standard output.

FORTRAN specifications:

ACMLVERSION (*MAJOR, MINOR, PATCH*) [SUBROUTINE]

MAJOR, *MINOR*, *PATCH* [INTEGER]

ACMLINFO () [SUBROUTINE]

C specifications:

`void` **acmlversion** (`int *major, int *minor, int *patch`); [function]

`void` **acmlinfo** (`void`); [function]

2.6 Library Documentation

The /Doc subdirectory of the top ACML installation directory, (e.g. /opt/acml2.6.0/Doc under Linux, or c:\Program Files\AMD\acml2.6.0\Doc under Windows), should contain this document in the following formats:

Printed Manual / PDF format – acml.pdf

Info Pages – acml.info (Linux only)

Html – html/index.html

Plain text – acml.txt

Under Linux the info file can be read using *info* after updating the environment variable INFOPATH to include the doc subdirectory of the ACML installation directory, e.g.

```
% setenv INFOPATH ${INFOPATH}:/opt/acml2.6.0/Doc

% info acml
```

or simply by using the full name of the file:

```
% info /opt/acml2.6.0/Doc/acml.info
```

2.7 Example programs calling ACML

The /examples subdirectory of the top ACML installation directory (for example, possible default locations are /opt/acml2.6.0/gnu64/examples under Linux, or, under windows, c:\Program Files\AMD\acml2.6.0\gnu32\examples), contains example programs showing how to call the ACML, along with a GNUmakefile to build and run them. Examples of calling both FORTRAN and C interfaces are included. They may be used as an ACML installation test.

Depending on where your copy of the ACML is installed, and which compiler and flags you wish to use, it may be necessary to modify some variables in the GNUmakefile before using it.

The 32-bit Windows versions of ACML assume that you have the Cygwin UNIX-like tools installed, and can use the *make* command that comes with them to build the examples.

For the 64-bit Windows version of ACML, it is not necessary to have the Cygwin tools. The examples directory contains a bat script, *acmlexample.bat*, which can be used to run one of the example programs. Another bat script, *acmlallexamples.bat*, builds and runs all the examples in the directory. Alternatively, if you do have the Cygwin tools installed, you can use the GNUmakefile to build the examples.

3 BLAS: Basic Linear Algebra Subprograms

The BLAS are a set of well defined basic linear algebra operations ([1], [2], [3]). These operations are subdivided into three groups:

- Level 1: operations acting on vectors only (e.g. dot product)
- Level 2: matrix-vector operations (e.g. matrix-vector multiplication)
- Level 3: matrix-matrix operations (e.g. matrix-matrix multiplication)

Efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The implementation of higher level linear algebra algorithms on these systems depends critically on the use of the BLAS as building blocks. AMD provides, as part of the ACML, an implementation of the BLAS optimized for performance on AMD64 processors.

For any information relating to the BLAS please refer to the BLAS FAQ:

http://www.netlib.org/blas/faq.html

ACML also includes interfaces to the extensions to Level 1 BLAS known as the sparse BLAS. These routines perform operations on a sparse vector x which is stored in compressed form and a vector y in full storage form. See reference [4] for more information.

4 LAPACK: Package of Linear Algebra Subroutines

4.1 Introduction to LAPACK

LAPACK ([5]) is a library of FORTRAN 77 subroutines for solving commonly occurring problems in numerical linear algebra. LAPACK components can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. Dense and banded matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.

LAPACK routines are written so that as much as possible of the computations is performed by calls to the BLAS. The efficiency of LAPACK routines depends, in large part, on the efficiency of the BLAS being called. Block algorithms are employed wherever possible to maximize the use of calls to level 3 BLAS, which generally run faster than lower level BLAS due to the high number of operations per memory access.

The performance of some of the LAPACK routines has been further improved by reworking the computational algorithms. Some of the LAPACK routines contained in ACML are therefore based on code that is different from the LAPACK sources available in the public domain. In all these cases the algorithmic and numerical properties of the original LAPACK routines have been strictly preserved. Furthermore, key LAPACK routines have been treated using OpenMP to take advantage of multiple processors when running on SMP machines. Your application will automatically benefit when you link with the OpenMP versions of ACML.

4.2 Reference sources for LAPACK

The LAPACK homepage can be accessed on the World Wide Web via the URL address:

http://www.netlib.org/lapack/

The on-line version of the Lapack User's Guide, Third Edition ([5]) is available from this homepage, or directly using the URL:

http://www.netlib.org/lapack/lug/index.html

The standard source code is available for download from netlib, with separate distributions for UNIX/Linux and Windows® installations:

http://www.netlib.org/lapack/lapack.tgz
http://www.netlib.org/lapack/lapack-pc.zip

A list of known problems, bugs, and compiler errors for LAPACK, as well as an errata list for the LAPACK User's Guide ([5]), is maintained on netlib

http://www.netlib.org/lapack/release_notes

A LAPACK FAQ (Frequently Asked Questions) file can also be accessed via the LAPACK homepage

http://www.netlib.org/lapack/faq.html

4.3 LAPACK block sizes and the ILAENV and ILAENVSET routines

As described in Section 6.2 of the LAPACK User's Guide, block sizes and other parameters used by various LAPACK routines are returned by the LAPACK inquiry function ILAENV. In ACML, values returned by ILAENV have been chosen to achieve very good performance on a wide variety of hardware and problem sizes.

In general it is unlikely that you will want or need to be concerned with these parameters. However, in some cases it may be that a default value returned by ILAENV is not optimal for your particular hardware and problem size. Following the advice in the LAPACK User's Guide may enable you to choose a better value in some circumstances.

For convenience, ACML includes a subroutine which allows you to override default values returned by ILAENV if you have superior knowledge. The routine is named ILAENVSET and has the following specification.

ILAENVSET [SUBROUTINE]
(*ISPEC,NAME,OPTS,N1,N2,N3,N4,NVALUE,INFO*)

`INTEGER` **ISPEC** [Input]
On input: *ISPEC* specifies the parameter to be set (see Section 6.2 of the LAPACK User's Guide for details).

`CHARACTER*(*)` **NAME** [Input]
On input: *NAME* specifies the name of the LAPACK subroutine for which the parameter is to be set.

`CHARACTER*(*)` **OPTS** [Input]
On input: *OPTS* is a character string of options to the subroutine.

`INTEGER` **N1, N2, N3, N4** [Input]
On input: *N1*, *N2*, *N3* and *N4* are problem dimensions. A value of -1 means that the dimension is unused or irrelevant.

`INTEGER` **NVALUE** [Input]
On input: *NVALUE* is the value to be set for the parameter specified by *ISPEC*. This value will be retrieved by any future call of `ILAENV` with similar arguments, including the call of `ILAENV` coming directly from the routine specified by argument *NAME*. In most cases, but not all, the value set will apply irrespective of the values of arguments *OPTS*, *N1*, *N2*, *N3* and *N4*.

`INTEGER` **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

All arguments of `ILAENVSET` apart from the last two, *NVALUE* and *INFO*, are identical to the arguments of `ILAENV`. `ILAENVSET` should be called *before* you call the LAPACK routine in question.

It should be noted that not all LAPACK routines make use of the `ILAENV` mechanism (because not all routines use blocked algorithms or require other tuning parameters). Calls of `ILAENVSET` with argument *NAME* set to the name of such a routine will fail with *INFO=0*.

In addition, the ACML versions of some important routines that do use blocked algorithms, such as the QR factorization routine `DGEQRF`, bypass `ILAENV` because they make use of a different tuning system which is independent of standard LAPACK. For all such routines, `ILAENVSET` can still be called with no error exit, but calls will have no effect on performance of the routine.

Below we give examples of how to call `ILAENVSET` in both FORTRAN and C.

Example (FORTRAN code):

```
      INTEGER ILO, IHI, INFO, N, NS
      CHARACTER COMPZ, JOB
      INTEGER ILAENV
      EXTERNAL ILAENV, ILAENVSET
      JOB = 'E'
      COMPZ = 'I'
      N = 512
      ILO = 1
      IHI = 512
C     Check the default shift parameter (ISPEC=4) used by DHSEQR
      NS = ILAENV(4, 'DHSEQR', JOB//COMPZ, N, ILO, IHI, -1)
      WRITE (*,*) 'Default NS = ', NS
C     Set a new value 5 for the shift parameter
      CALL ILAENVSET(4, 'DHSEQR', JOB//COMPZ, N, ILO, IHI, -1, 5, INFO)
C     Then check the shift parameter again
      NS = ILAENV(4, 'DHSEQR', JOB//COMPZ, N, ILO, IHI, -1)
      WRITE (*,*) 'Revised NS = ', NS
      END
```

Example (C code):

```
#include <acml.h>
#include <stdio.h>
int main(void)
{
  int n=512, ilo=1, ihi=512, ns, info;
  char compz = 'I', job = 'E', opts[3];
  opts[0] = job;
  opts[1] = compz;
  opts[2] = '\0';
/* Check the default shift parameter (ISPEC=4) used by DHSEQR */
  ns = ilaenv(4, "DHSEQR", opts, n, ilo, ihi, -1);
  printf("Default ns = %d\n", ns);
/* Set a new value 5 for the shift parameter */
  ilaenvset(4, "DHSEQR", opts, n, ilo, ihi, -1, 5, &info);
/* Then check the shift parameter again */
  ns = ilaenv(4, "DHSEQR", opts, n, ilo, ihi, -1);
  printf("Revised ns = %d\n", ns);
  return 0;
}
```

5 Fast Fourier Transforms (FFTs)

5.1 Introduction to FFTs

5.1.1 Data Types and Storage

There are two main types of DFTs:

- routines for the transformation of complex data: in the ACML, these routines have names beginning with `ZFFT` or `CFFT`, for double and single precision, respectively;
- routines for the transformation of real to complex data and vice versa: in the ACML the names for the former begin with `DZFFT` or `SCFFT`, for double and single precision, respectively; the names for the latter begin with `ZDFFT` or `CSFFT`.

Complex data

The simplest transforms to describe are those performed on sequences of complex data. Such data are stored as arrays of type complex. The result of a complex FFT is also a complex sequence of the same length and, for the simple interfaces, is written back to the original array. Where multiple (m, say), same-length sequences (of length n) of complex data are to be transformed, the sequences are held in a single complex array; in the simple interfaces the array will be of length $m * n$ containing m end-to-end sequences and the results of the m FFTs are returned in the original array. Expert interfaces are provided which give: greater flexibility in the storage of the original data and results, user provided scaling, and whether results should be written to a separate array or not.

The definition of a complex DFT used here is given by:

$$\tilde{x}_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} x_k \exp\left(\pm i \frac{2\pi jk}{n}\right) \text{ for } j = 0, 1, \ldots, n-1$$

where x_k are the complex data to be transformed, $\tilde{x}_j$ are the transformed data, and the sign of $\pm$ determines the direction of the transform: $(-)$ for forward and $(+)$ for backward. Note that, in this definition, both directional transforms have the same scaling and performing both consecutively recovers the original data; this is the prescribed scaling provided in the simple FFT interfaces, whereas, in the expert interfaces, the scaling factor must be supplied by the user.

For the simple interfaces, a two dimensional array of complex data, with m rows and n columns is stored in the same order as a set of n sequences of length m (as described above). That is, column elements are stored contiguously and the first element of the next column follows the last element of the current column. In the expert interfaces, column elements may be separated by a fixed step length (increment) while row elements may be separated by a second increment; if the first increment is 1 and the second increment is m then we have the same storage as in the simple interface.

The definition of a complex 2D DFT used here is given by:

$$\tilde{x}_{jp} = \frac{1}{\sqrt{m*n}} \sum_{l=0}^{m-1} \sum_{k=0}^{n-1} x_{kl} \exp\left(\pm i \frac{2\pi jk}{n}\right) \exp\left(\pm i \frac{2\pi pl}{m}\right)$$

for $j = 0, 1, \ldots, n-1$ and $l = 0, 1, \ldots, m-1$, where x_{kl} are the complex data to be transformed, $\tilde{x}_{jp}$ are the transformed data, and the sign of $\pm$ determines the direction of the transform.

Real data

The DFT of a sequence of real data results in a special form of complex sequence known as a Hermitian sequence. The symmetries defining such a sequence mean that it can be fully represented by a set of n real values, where n is the length of the original real sequence. It is therefore conventional for the array containing the real sequence to be overwritten by such a representation of the transformed Hermitian sequence.

If the original sequence is purely real valued, i.e. $z_j = x_j$, then the definition of the real DFT used here is given by:

$$\tilde{z}_j = a_j + ib_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} x_k \exp\left(-i \frac{2\pi jk}{n}\right) \text{ for } j = 0, 1, \ldots, n-1$$

where x_k are the real data to be transformed, $\tilde{z}_j$ are the transformed complex data.

In full complex representation, the Hermitian sequence would be a sequence of n complex values $Z(i)$ for $i = 0, 1, ..., n-1$, where $Z(n-j)$ is the complex conjugate of $Z(j)$ for $j = 1, 2, ..., (n-1)/2$; $Z(0)$ is real valued; and, if n is even, $Z(n/2)$ is real valued. In ACML, the representation of Hermitian sequences used on output from `DZFFT` routines and on input to `ZDFFT` routines is as follows:
let X be an array of length N and with first index 0,

- $X(i)$ contains the real part of $Z(i)$ for $i = 0, ..., N/2$
- $X(N-i)$ contains the imaginary part of $Z(i)$ for $i = 1, ..., (N-1)/2$

Also, given a Hermitian sequence, the discrete transform can be written as:

$$x_j = \frac{1}{\sqrt{n}} \left(a_0 + 2 \sum_{k=1}^{n/2-1} \left(a_k \cos\left(\frac{2\pi jk}{n}\right) - b_k \sin\left(\frac{2\pi jk}{n}\right)\right) + a_{n/2} \right)$$

where $a_{n/2} = 0$ if n is odd, and $\tilde{z}_k = a_k + ib_k$ is the Hermitian sequence to be transformed. Note that, in the above definitions, both transforms have the same (negative) sign in the exponent; performing both consecutively does not recover the original data. To recover original real data, or otherwise to perform an inverse transform on a set of Hermitian data, the Hermitian data must be conjugated prior to performing the transform (i.e. changing the sign of the stored imaginary parts).

5.1.2 Efficiency

The efficiency of the FFT is maximized by choosing the sequence length to be a power of 2. Good efficiency can also be achieved when the sequence length has small prime factors, up to a factor 13; however, the time taken for an FFT increases as the size of the prime factor increases.

5.2 FFTs on Complex Sequences

5.2.1 FFT of a single sequence

The routines documented here compute the discrete Fourier transform (DFT) of a sequence of complex numbers in either single or double precision arithmetic. The DFT is computed using a highly-efficient FFT algorithm. There are two sets of interfaces available: simple drivers and expert drivers. The simple drivers perform in-place transforms on data held contiguously in memory using a fixed scaling factor; these are simpler to use and are sufficient for many problems. The expert drivers offer greater flexibility by including a number of additional arguments. These allow you to control: the scaling factor applied; whether the result should be output to a separate vector; and, the increments used in storing successive elements of both the input sequence and the result.

`ZFFT1D` Routine Documentation

ZFFT1D (*MODE,N,X,COMM,INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]

The value of *MODE* on input determines the operation performed by `ZFFT1D`. On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to `ZFFT1D`.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to `ZFFT1D`.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

`INTEGER` **N** [Input]

On input: N is the length of the complex sequence X

`COMPLEX*16` **X(N)** [Input/Output]

On input: X contains the complex sequence of length N to be transformed.
On output: X contains the transformed sequence.

`COMPLEX*16` **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL ZFFT1D(0,N,X,COMM,INFO)
      CALL ZFFT1D(-1,N,X,COMM,INFO)
      CALL ZFFT1D(-1,N,Y,COMM,INFO)
      DO 10 I = 1, N
         X(I) = X(I)*DCONJG(Y(I))
10    CONTINUE
      CALL ZFFT1D(1,N,X,COMM,INFO)
```

CFFT1D Routine Documentation

CFFT1D (*MODE,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by CFFT1D. On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1D.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1D.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

INTEGER **N** [Input]

On input: N is the length of the complex sequence X

COMPLEX **X(N)** [Input/Output]

On input: X contains the complex sequence of length N to be transformed. On output: X contains the transformed sequence.

COMPLEX **COMM(5*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT1D(0,N,X,COMM,INFO)
      CALL CFFT1D(-1,N,X,COMM,INFO)
      CALL CFFT1D(-1,N,Y,COMM,INFO)
      DO 10 I = 1, N
         X(I) = X(I)*CONJG(Y(I))
10    CONTINUE
      CALL CFFT1D(1,N,X,COMM,INFO)
```

ZFFT1DX Routine Documentation

ZFFT1DX (*MODE,SCALE,INPL,N,X,INCX,Y,INCY,COMM, INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]

The value of *MODE* on input determines the operation performed by `ZFFT1DX`. On input:

MODE=0 : only initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to `ZFFT1DX`.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to `ZFFT1DX`.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

`DOUBLE PRECISION` **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequence

`LOGICAL` **INPL** [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequence; otherwise the output sequence is returned in *Y*.

`INTEGER` **N** [Input]

On input: *N* is the number of elements to be transformed

`COMPLEX*16` **X(1+(N-1)*INCX)** [Input/Output]

On input: *X* contains the complex sequence of length *N* to be transformed, with the ith element stored in X(1+(i-1)*INCX).
On output: if *INPL* is .TRUE. then *X* contains the transformed sequence in the same locations as on input; otherwise *X* remains unchanged.

`INTEGER` **INCX** [Input]

On input: *INCX* is the increment used to store successive elements of a sequence in *X*.
Constraint: *INCX* > 0.

`COMPLEX*16` **Y(1+(N-1)*INCY)** [Output]

On output: if *INPL* is .FALSE. then *Y* contains the transformed sequence, with the ith element stored in Y(1+(i-1)*INCY); otherwise *Y* is not referenced.

`INTEGER` **INCY** [Input]

On input: *INCY* is the increment used to store successive elements of a sequence in *Y*. If *INPL* is .TRUE. then *INCY* is not referenced.
Constraint: *INCY* > 0.

`COMPLEX*16` **COMM(5*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

`Example:`

```
C     Forward FFTs are performed unscaled and in-place on contiguous
C     vectors X and Y following initialisation. Manipulations on
C     resultant Fourier coefficients are stored in X which is then
C     transformed back.
C
        SCALE = 1.0D0
        INPL = .TRUE.
        CALL ZFFT1DX(0,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL ZFFT1DX(-1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL ZFFT1DX(-1,SCALE,INPL,N,Y,1,DUM,1,COMM,INFO)
        DO 10 I = 1, N
           X(I) = X(I)*DCONJG(Y(I))/DBLE(N)
   10   CONTINUE
        CALL ZFFT1DX(1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
```

`CFFT1DX` Routine Documentation

CFFT1DX (*MODE,SCALE,INPL,N,X,INCX,Y,INCY,COMM, INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]

The value of *MODE* on input determines the operation performed by `CFFT1DX`.
On input:

MODE=0 : only initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to `CFFT1DX`.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to `CFFT1DX`.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

`REAL` **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequence

`LOGICAL` **INPL** [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequence; otherwise the output sequence is returned in *Y*.

`INTEGER` **N** [Input]

On input: *N* is the number of elements to be transformed

`COMPLEX` **X(1+(N-1)*INCX)** [Input/Output]

On input: *X* contains the complex sequence of length *N* to be transformed, with the ith element stored in X(1+(i-1)*INCX).
On output: if *INPL* is .TRUE. then *X* contains the transformed sequence in the same locations as on input; otherwise *X* remains unchanged.

`INTEGER` **INCX** [Input]

On input: *INCX* is the increment used to store successive elements of a sequence in *X*.
Constraint: *INCX* > 0.

`COMPLEX` **Y(1+(N-1)*INCY)** [Output]

On output: if *INPL* is .FALSE. then *Y* contains the transformed sequence, with the ith element stored in Y(1+(i-1)*INCY); otherwise *Y* is not referenced.

`INTEGER` **INCY** [Input]

On input: *INCY* is the increment used to store successive elements of a sequence in *Y*. If *INPL* is .TRUE. then *INCY* is not referenced.
Constraint: *INCY* > 0.

COMPLEX **COMM(5*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on contiguous
C     vectors X and Y following initialisation. Manipulations on
C     resultant Fourier coefficients are stored in X which is then
C     transformed back.
C
        SCALE = 1.0
        INPL = .TRUE.
        CALL CFFT1DX(0,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL CFFT1DX(-1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL CFFT1DX(-1,SCALE,INPL,N,Y,1,DUM,1,COMM,INFO)
        DO 10 I = 1, N
           X(I) = X(I)*CONJG(Y(I))/REAL(N)
   10   CONTINUE
        CALL CFFT1DX(1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
```

5.2.2 FFT of multiple complex sequences

The routines documented here compute the discrete Fourier transforms (DFTs) of a number of sequences of complex numbers in either single or double precision arithmetic. The sequences must all have the same length. The DFTs are computed using a highly-efficient FFT algorithm. There are two sets of interfaces available: simple drivers and expert drivers. The simple drivers perform in-place transforms on data held contiguously in memory using a fixed scaling factor; these are simpler to use and are sufficient for many problems. The expert drivers offer greater flexibility by including a number of additional arguments. These allow you to control: the scaling factor applied; whether the result should be output to a separate vector; the increments used in storing successive elements of a given sequence (for both input and output sequences); and the increments used in storing corresponding elements in successive sequences (for both input and output).

`ZFFT1M` Routine Documentation

ZFFT1M (*MODE,M,N,X,COMM,INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]

The value of *MODE* on input determines the operation performed by `ZFFT1M`.
On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : forward transforms are performed. Initializations are assumed to have been performed by a prior call to `ZFFT1M`.

MODE=1 : backward (reverse) transforms are performed. Initializations are assumed to have been performed by a prior call to `ZFFT1M`.

MODE=−2 : initializations and forward transforms are performed.

MODE=2 : initializations and backward transforms are performed.

`INTEGER` **M** [Input]

On input: M is the number of sequences to be transformed.

`INTEGER` **N** [Input]

On input: N is the length of the complex sequences in X

`COMPLEX*16` **X(N*M)** [Input/Output]

On input: X contains the M complex sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed sequences.

`COMPLEX*16` **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL ZFFT1M(0,1,N,X,COMM,INFO)
      CALL ZFFT1M(-1,2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*DCONJG(X(I,2))
         X(I,2) = DCMPLX(0.0D0,1.0D0)*X(I,2)
10    CONTINUE
      CALL ZFFT1M(1,2,N,X(1,2),COMM,INFO)
```

CFFT1M Routine Documentation

CFFT1M (*MODE,M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by CFFT1M.
On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : forward transforms are performed. Initializations are assumed to have been performed by a prior call to CFFT1M.

MODE=1 : backward (reverse) transforms are performed. Initializations are assumed to have been performed by a prior call to CFFT1M.

MODE=−2 : initializations and forward transforms are performed.

MODE=2 : initializations and backward transforms are performed.

INTEGER **M** [Input]

On input: M is the number of sequences to be transformed.

INTEGER **N** [Input]

On input: N is the length of the complex sequences in X

COMPLEX **X(N*M)** [Input/Output]

On input: X contains the M complex sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed sequences.

COMPLEX **COMM(5*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT1M(0,1,N,X,COMM,INFO)
      CALL CFFT1M(-1,2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*CONJG(X(I,2))
         X(I,2) = CMPLX(0.0D0,1.0D0)*X(I,2)
10    CONTINUE
      CALL CFFT1M(1,2,N,X(1,2),COMM,INFO)
```

ZFFT1MX Routine Documentation

ZFFT1MX (*MODE,SCALE,INPL,NSEQ,N,X,INCX1,INCX2, Y,INCY1,INCY2,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by ZFFT1MX. On input:

MODE=0 : only initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1MX.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1MX.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

DOUBLE PRECISION **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL **INPL** [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequences; otherwise the output sequences are returned in *Y*.

INTEGER **NSEQ** [Input]

On input: *NSEQ* is the number of sequences to be transformed

INTEGER **N** [Input]

On input: *N* is the number of elements in each sequence to be transformed

COMPLEX*16 **X(1+(N-1)*INCX1+(NSEQ-1)*INCX2)** [Input/Output]

On input: *X* contains the *NSEQ* complex sequences of length *N* to be transformed; the ith element of sequence j is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).

On output: if *INPL* is .TRUE. then *X* contains the transformed sequences in the same locations as on input; otherwise *X* remains unchanged.

INTEGER **INCX1** [Input]

On input: *INCX1* is the increment used to store successive elements of a given sequence in *X* (*INCX1*=1 for contiguous data).

Constraint: *INCX1* > 0.

INTEGER **INCX2** [Input]

On input: *INCX2* is the increment used to store corresponding elements of successive sequences in *X* (*INCX2*=*N* for contiguous data).

Constraint: *INCX2* > 0.

COMPLEX*16 **Y(1+(N-1)*INCY1+(NSEQ-1)*INCY2)** [Output]
On output: if *INPL* is .FALSE. then *Y* contains the transformed sequences with the ith element of sequence j stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); otherwise *Y* is not referenced.

INTEGER **INCY1** [Input]
On input: *INCY1* is the increment used to store successive elements of a given sequence in *Y*. If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

INTEGER **INCY2** [Input]
On input: *INCY2* is the increment used to store corresponding elements of successive sequences in *Y* (*INCY2*=*N* for contiguous data). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraint: *INCY2* > 0.

COMPLEX*16 **COMM(3*N+100)** [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on two
C     contiguous vectors stored in the first two columns of X.
C     Manipulations are stored in 2nd and 3rd columns of X which are
C     then transformed back.
C
        COMPLEX *16 X(N,3)
        SCALE = 1.0D0
        INPL = .TRUE.
        CALL ZFFT1MX(0,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        CALL ZFFT1MX(-1,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        DO 10 I = 1, N
           X(I,3) = X(I,1)*DCONJG(X(I,2))/DBLE(N)
           X(I,2) = DCMPLX(0.0D0,1.0D0)*X(I,2)/DBLE(N)
   10   CONTINUE
        CALL ZFFT1MX(1,SCALE,INPL,2,N,X(1,2),1,N,DUM,1,N,COMM,INFO)
```

CFFT1MX Routine Documentation

CFFT1MX (*MODE,SCALE,INPL,NSEQ,N,X,INCX1,INCX2, Y,INCY1,INCY2,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by CFFT1MX. On input:

MODE=0 : only initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1MX.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1MX.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

REAL **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL **INPL** [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequences; otherwise the output sequences are returned in *Y*.

INTEGER **NSEQ** [Input]

On input: *NSEQ* is the number of sequences to be transformed

INTEGER **N** [Input]

On input: *N* is the number of elements in each sequence to be transformed

COMPLEX **X(1+(N-1)*INCX1+(NSEQ-1)*INCX2)** [Input/Output]

On input: *X* contains the *NSEQ* complex sequences of length *N* to be transformed; the ith element of sequence j is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).

On output: if *INPL* is .TRUE. then *X* contains the transformed sequences in the same locations as on input; otherwise *X* remains unchanged.

INTEGER **INCX1** [Input]

On input: *INCX1* is the increment used to store successive elements of a given sequence in *X* (*INCX1*=1 for contiguous data).

Constraint: *INCX1* > 0.

INTEGER **INCX2** [Input]

On input: *INCX2* is the increment used to store corresponding elements of successive sequences in *X* (*INCX2*=*N* for contiguous data).

Constraint: *INCX2* > 0.

COMPLEX **Y(1+(N-1)*INCY1+(NSEQ-1)*INCY2)** [Output]
On output: if *INPL* is .FALSE. then *Y* contains the transformed sequences with the ith element of sequence j stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); otherwise *Y* is not referenced.

INTEGER **INCY1** [Input]
On input: *INCY1* is the increment used to store successive elements of a given sequence in *Y*. If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

INTEGER **INCY2** [Input]
On input: *INCY2* is the increment used to store corresponding elements of successive sequences in *Y* (*INCY2*=*N* for contiguous data). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraint: *INCY2* > 0.

COMPLEX **COMM(5*N+100)** [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on two
C     contiguous vectors stored in the first two columns of X.
C     Manipulations are stored in 2nd and 3rd columns of X which are
C     then transformed back.
C
        COMPLEX X(N,3)
        SCALE = 1.0
        INPL = .TRUE.
        CALL CFFT1MX(0,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        CALL CFFT1MX(-1,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        DO 10 I = 1, N
           X(I,3) = X(I,1)*CONJG(X(I,2))/REAL(N)
           X(I,2) = CMPLX(0.0D0,1.0D0)*X(I,2)/REAL(N)
   10   CONTINUE
        CALL CFFT1MX(1,SCALE,INPL,2,N,X(1,2),1,N,DUM,1,N,COMM,INFO)
```

5.2.3 2D FFT of two-dimensional arrays of data

The routines documented here compute the two-dimensional discrete Fourier transforms (DFT) of a two-dimensional array of complex numbers in either single or double precision arithmetic. The 2D DFT is computed using a highly-efficient FFT algorithm.
There are two sets of interfaces available: simple drivers and expert drivers. The simple drivers perform in-place transforms on data held contiguously in memory using a fixed scaling factor; these are simpler to use and are sufficient for many problems. The expert drivers offer greater flexibility by including a number of additional arguments. These allow you to control: the scaling factor applied; whether the result should be output to a separate array; the increments used in storing successive elements in each dimension (for both input and output); and the facility to not perform a final transposition. This final facility is useful for those cases where a forward and backward transform are to be applied with some data manipulations in between; here two whole transpositions can be saved.

`ZFFT2D` Routine Documentation

ZFFT2D (*MODE,M,N,X,COMM,INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]
The value of *MODE* on input determines the direction of transform to be performed by `ZFFT2D`.
On input:

MODE=−1 : forward 2D transform is performed.

MODE=1 : backward (reverse) 2D transform is performed.

`INTEGER` **M** [Input]
On input: M is the number of rows in the 2D array of data to be transformed. If X is declared as a 2D array then M is the first dimension of X.

`INTEGER` **N** [Input]
On input: N is the number of columns in the 2D array of data to be transformed. If X is declared as a 2D array then M is the second dimension of X.

`COMPLEX*16` **X(M*N)** [Input/Output]
On input: X contains the M by N complex 2D array to be transformed. Element ij is stored in location $i + (j-1) * M$ of X.
On output: X contains the transformed sequence.

`COMPLEX*16` **COMM(M*N+3*(M+N))** [Input/Output]
COMM is a communication array used as temporary store.

`INTEGER` **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

`Example:`

```
      CALL ZFFT2D(-1,M,N,X,COMM,INFO)
      DO 20 J = 1, N
         DO 10 I = 1, MIN(J-1,M)
            X(I,J) = 0.5D0*(X(I,J) + X(J,I))
            X(J,I) = DCONJG(X(I,J))
10       CONTINUE
20    CONTINUE
      CALL ZFFT2D(1,M,N,X,COMM,INFO)
```

CFFT2D Routine Documentation

CFFT2D (*MODE,M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the direction of transform to be performed by ZFFT2D.

On input:

MODE=−1 : forward 2D transform is performed.

MODE=1 : backward (reverse) 2D transform is performed.

INTEGER **M** [Input]

On input: M is the number of rows in the 2D array of data to be transformed. If X is declared as a 2D array then M is the first dimension of X.

INTEGER **N** [Input]

On input: N is the number of columns in the 2D array of data to be transformed. If X is declared as a 2D array then M is the second dimension of X.

COMPLEX **X(M*N)** [Input/Output]

On input: X contains the M by N complex 2D array to be transformed. Element ij is stored in location $i + (j-1) * M$ of X.

On output: X contains the transformed sequence.

COMPLEX **COMM(M*N+5*(M+N))** [Input/Output]

COMM is a communication array used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT2D(-1,M,N,X,COMM,INFO)
      DO 20 J = 1, N
         DO 10 I = 1, MIN(J-1,M)
            X(I,J) = 0.5D0*(X(I,J) + X(J,I))
            X(J,I) = CONJG(X(I,J))
10       CONTINUE
20    CONTINUE
      CALL CFFT2D(1,M,N,X,COMM,INFO)
```

ZFFT2DX Routine Documentation

ZFFT2DX (*MODE,SCALE,LTRANS,INPL,M,N,X,INCX1, INCX2,Y,INCY1,INCY2,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by ZFFT2DX.
On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT2DX.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT2DX.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

DOUBLE PRECISION **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL **LTRANS** [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data consistent with the values for arguments *INPL*, *INCX1*, *INCX2*, *INCY1* and *INCY2*. If *LTRANS* is .FALSE. then the final transposition is not performed explicitly; the storage format on output is determined by whether the output data is stored contiguously or not – please see the output specifications for X and Y for details.

LOGICAL **INPL** [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER **M** [Input]

On input: M is the first dimension of the 2D transform.

INTEGER **N** [Input]

On input: N is the second dimension of the 2D transform.

COMPLEX*16 **X(1+(M-1)*INCX1+(N-1)*INCX2)** [Input/Output]

On input: X contains the M by N complex 2D data array to be transformed; the (ij)th element is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).
On output: if *INPL* is .TRUE. then X contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; in locations X((i-1)*N+j) when *LTRANS*=.FALSE., *INCX1*=1 and *INCX2*=M; and otherwise in the same locations as on input. If *INPL* is .FALSE. X remains unchanged.

INTEGER **INCX1** [Input]

On input: *INCX1* is the increment used to store, in X, successive elements in the first dimension (*INCX1*=1 for contiguous data).
Constraint: *INCX1* > 0.

`INTEGER` **INCX2** [Input]

On input: *INCX2* is the increment used to store, in *X*, successive elements in the second dimension (*INCX2*=M for contiguous data).
Constraint: *INCX2* > 0.

`COMPLEX*16` **Y(1+(M-1)*INCY1+(N-1)*INCY2)** [Output]

On output: if *INPL* is .FALSE. then *Y* contains the transformed data. If *LTRANS*=.TRUE. then the (ij)th data element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); if *LTRANS*=.FALSE., *INCY1*=1 and *INCY2*=M then the (ij)th data element is stored in Y((i-1)*N+j); and otherwise the (ij)th element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2). If *INPL* is .TRUE. then *Y* is not referenced.

`INTEGER` **INCY1** [Input]

On input: *INCY1* is the increment used to store successive elements in the first dimension in *Y* (*INCY1*=1 for contiguous data). If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

`INTEGER` **INCY2** [Input]

On input: *INCY2* is the increment used to store successive elements in the second dimension in *Y* (*INCY2*=*M* for contiguous data). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraint: *INCY2* > 0.

`COMPLEX*16` **COMM(M*N+3*M+3*N+200)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same dimensions *M* and *N*. The remainder is used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

`Example:`

```
C     Forward 2D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        COMPLEX *16 X(LDX,N), Y(M*N)
        SCALE = 1.0D0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL ZFFT2DX(0,SCALE,LTRANS,INPL,M,N,X,1,LDX,Y,1,M,COMM,INFO)
        CALL ZFFT2DX(-1,SCALE,LTRANS,INPL,M,N,X,1,LDX,Y,1,M,COMM,INFO)
        IY = 1
        DO 20 I = M
           DO 10 J = 1, N
              Y(IY) = 0.5D0*Y(IY)*EXP(0.001D0*(I+J-2))
              IY = IY + 1
   10      CONTINUE
   20   CONTINUE
        SCALE = 1.0D0/DBLE(M*N)
        CALL ZFFT2DX(1,SCALE,LTRANS,INPL,N,M,Y,1,N,X,1,LDX,COMM,INFO)
```

CFFT2DX Routine Documentation

CFFT2DX (*MODE,SCALE,LTRANS,INPL,M,N,X,INCX1, INCX2,Y,INCY1,INCY2,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by CFFT2DX.
On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.

MODE=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CFFT2DX.

MODE=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to CFFT2DX.

MODE=−2 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

REAL **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL **LTRANS** [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data consistent with the values for arguments *INPL*, *INCX1*, *INCX2*, *INCY1* and *INCY2*. If *LTRANS* is .FALSE. then the final transposition is not performed explicitly; the storage format on output is determined by whether the output data is stored contiguously or not – please see the output specifications for X and Y for details.

LOGICAL **INPL** [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER **M** [Input]

On input: M is the first dimension of the 2D transform.

INTEGER **N** [Input]

On input: N is the second dimension of the 2D transform.

COMPLEX **X(1+(M-1)*INCX1+(N-1)*INCX2)** [Input/Output]

On input: X contains the M by N complex 2D data array to be transformed; the (ij)th element is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).
On output: if *INPL* is .TRUE. then X contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; in locations X((i-1)*N+j) when *LTRANS*=.FALSE., *INCX1*=1 and *INCX2*=M; and otherwise in the same locations as on input. If *INPL* is .FALSE. X remains unchanged.

INTEGER **INCX1** [Input]

On input: *INCX1* is the increment used to store, in X, successive elements in the first dimension (*INCX1*=1 for contiguous data).
Constraint: *INCX1* > 0.

`INTEGER` **INCX2** [Input]

On input: *INCX2* is the increment used to store, in *X*, successive elements in the second dimension (*INCX2*=M for contiguous data).
Constraint: *INCX2* > 0.

`COMPLEX` **Y(1+(M-1)*INCY1+(N-1)*INCY2)** [Output]

On output: if *INPL* is .FALSE. then *Y* contains the transformed data. If *LTRANS*=.TRUE. then the (ij)th data element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); if *LTRANS*=.FALSE., *INCY1*=1 and *INCY2*=M then the (ij)th data element is stored in Y((i-1)*N+j); and otherwise the (ij)th element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2). If *INPL* is .TRUE. then *Y* is not referenced.

`INTEGER` **INCY1** [Input]

On input: *INCY1* is the increment used to store successive elements in the first dimension in *Y* (*INCY1*=1 for contiguous data). If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

`INTEGER` **INCY2** [Input]

On input: *INCY2* is the increment used to store successive elements in the second dimension in *Y* (*INCY2*=*M* for contiguous data). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraint: *INCY2* > 0.

`COMPLEX` **COMM(M*N+5*M+5*N+200)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same dimensions *M* and *N*. The remainder is used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

`Example:`

```
C     Forward 2D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        COMPLEX X(LDX,N), Y(M*N)
        SCALE = 1.0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL CFFT2DX(0,SCALE,LTRANS,INPL,M,N,X,1,LDX,Y,1,M,COMM,INFO)
        CALL CFFT2DX(-1,SCALE,LTRANS,INPL,M,N,X,1,LDX,Y,1,M,COMM,INFO)
        IY = 1
        DO 20 I = M
           DO 10 J = 1, N
              Y(IY) = 0.5*Y(IY)*EXP(-0.001*REAL(I+J-2))
              IY = IY + 1
   10      CONTINUE
   20   CONTINUE
        SCALE = 1.0/REAL(M*N)
        CALL CFFT2DX(1,SCALE,LTRANS,INPL,N,M,Y,1,N,X,1,LDX,COMM,INFO)
```

5.2.4 3D FFT of three-dimensional arrays of data

The routines documented here compute the three-dimensional discrete Fourier transforms (DFT) of a three-dimensional array of complex numbers in either single or double precision arithmetic. The 3D DFT is computed using a highly-efficient FFT algorithm.

`ZFFT3D` Routine Documentation

ZFFT3D (*MODE,L,M,N,X,COMM,INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]

The value of *MODE* on input determines the direction of transform to be performed by `ZFFT3D`.

On input:

MODE=−1 : forward 3D transform is performed.

MODE=1 : backward (reverse) 3D transform is performed.

`INTEGER` **L** [Input]

On input: the length of the first dimension of the 3D array of data to be transformed. If *X* is declared as a 3D array then *L* is the first dimension of *X*.

`INTEGER` **M** [Input]

On input: the length of the second dimension of the 3D array of data to be transformed. If *X* is declared as a 3D array then *M* is the second dimension of *X*.

`INTEGER` **N** [Input]

On input: the length of the third dimension of the 3D array of data to be transformed. If *X* is declared as a 3D array then *N* is the third dimension of *X*.

`COMPLEX*16` **X(L*M*N)** [Input/Output]

On input: *X* contains the *L* by *M* by *N* complex 3D array to be transformed. Element ijk is stored in location $i + (j-1)*L + (k-1)*L*M$ of *X*.

On output: *X* contains the transformed sequence.

`COMPLEX*16` **COMM(L*M*N+3*(L+M+N))** [Input/Output]

COMM is a communication array used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL ZFFT3D(-1,L,M,N,X,COMM,INFO)
      DO 30 K = 1, N
        DO 20 J = 1, M
          DO 10 I = 1, L
            X(I,J) = X(I,J)*EXP(-0.001D0*DBLE(I+J+K))
10        CONTINUE
20      CONTINUE
30    CONTINUE
      CALL ZFFT3D(1,L,M,N,X,COMM,INFO)
```

CFFT3D Routine Documentation

CFFT3D (*MODE,L,M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the direction of transform to be performed by CFFT3D.

On input:

MODE=−1 : forward 3D transform is performed.

MODE=1 : backward (reverse) 3D transform is performed.

INTEGER **L** [Input]

On input: the length of the first dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the first dimension of X.

INTEGER **M** [Input]

On input: the length of the second dimension of the 3D array of data to be transformed. If X is declared as a 3D array then M is the second dimension of X.

INTEGER **N** [Input]

On input: the length of the third dimension of the 3D array of data to be transformed. If X is declared as a 3D array then N is the third dimension of X.

COMPLEX **X(L*M*N)** [Input/Output]

On input: X contains the L by M by N complex 3D array to be transformed. Element ijk is stored in location $i + (j-1) * L + (k-1) * L * M$ of X.

On output: X contains the transformed sequence.

COMPLEX **COMM(L*M*N+5*(L+M+N))** [Input/Output]

COMM is a communication array used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT3D(-1,L,M,N,X,COMM,INFO)
      DO 30 K = 1, N
        DO 20 J = 1, M
          DO 10 I = 1, L
            X(I,J) = X(I,J)*EXP(-0.001D0*REAL(I+J+K))
10        CONTINUE
20      CONTINUE
30    CONTINUE
      CALL CFFT3D(1,L,M,N,X,COMM,INFO)
```

ZFFT3DX Routine Documentation

ZFFT3DX (*MODE,SCALE,LTRANS,INPL,L,M,N,X,Y, COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by ZFFT3DX. On input:

MODE=−1 : a forward transform is performed.

MODE=1 : a backward (reverse) transform is performed.

DOUBLE PRECISION **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL **LTRANS** [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data using the same storage format as the input data. If *LTRANS* is .FALSE. then the final transposition is not performed and transformed data is stored, in *X* or *Y*, in transposed form.

LOGICAL **INPL** [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequences; otherwise the output sequences are returned in *Y*.

INTEGER **L** [Input]

On input: *L* is the first dimension of the 3D transform.

INTEGER **M** [Input]

On input: *M* is the second dimension of the 3D transform.

INTEGER **N** [Input]

On input: *N* is the third dimension of the 3D transform.

COMPLEX*16 **X(L*M*N)** [Input/Output]

On input: *X* contains the *L* by *M* by *N* complex 3D data array to be transformed; the (ijk)th element is stored in X(i+(j-1)*L+(k-1)*L*M).

On output: if *INPL* is .TRUE. then *X* contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; or in locations X(k+(j-1)*N+(i-1)*N*M) when *LTRANS*=.FALSE. If *INPL* is .FALSE. *X* remains unchanged.

COMPLEX*16 **Y(L*M*N)** [Output]

On output: if *INPL* is .FALSE. then *Y* contains the three-dimensional transformed data. If *LTRANS*=.TRUE. then the (ijk)th data element is stored in Y(i+(j-1)*L+(k-1)*L*M); otherwise, the (ijk)th data element is stored in Y(k+(j-1)*N+(i-1)*N*M). If *INPL* is .TRUE. then *Y* is not referenced.

COMPLEX*16 **COMM(L*M*N+5*(L+M+N)+200)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence dimensions. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 3D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        COMPLEX *16 X(L*M*N), Y(L*M*N)
        SCALE = 1.0D0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL ZFFT3DX(-1,SCALE,LTRANS,INPL,L,M,N,X,Y,COMM,INFO)
        IY = 1
        DO 20 I = 1, L
           DO 40 J = 1, M
              DO 10 K = 1, N
                 Y(IY) = Y(IY)*EXP(-0.001D0*DBLE(I+J+K-3))
                 IY = IY + 1
   10         CONTINUE
   20      CONTINUE
           SCALE = 1.0D0/DBLE(L*M*N)
           CALL ZFFT3DX(1,SCALE,LTRANS,INPL,N,M,L,Y,X,COMM,INFO)
```

CFFT3DX Routine Documentation

CFFT3DX (*MODE,SCALE,LTRANS,INPL,L,M,N,X,Y, COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by CFFT3DX.
On input:

MODE=−1 : a forward transform is performed.

MODE=1 : a backward (reverse) transform is performed.

REAL **SCALE** [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL **LTRANS** [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data using the same storage format as the input data. If *LTRANS* is .FALSE. then the final transposition is not performed and transformed data is stored, in *X* or *Y*, in transposed form.

LOGICAL **INPL** [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequences; otherwise the output sequences are returned in *Y*.

INTEGER **L** [Input]

On input: *L* is the first dimension of the 3D transform.

INTEGER **M** [Input]

On input: *M* is the second dimension of the 3D transform.

INTEGER **N** [Input]

On input: *N* is the third dimension of the 3D transform.

COMPLEX **X(L*M*N)** [Input/Output]

On input: *X* contains the *L* by *M* by *N* complex 3D data array to be transformed; the (ijk)th element is stored in X(i+(j-1)*L+(k-1)*L*M).
On output: if *INPL* is .TRUE. then *X* contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; or in locations X(k+(j-1)*N+(i-1)*N*M) when *LTRANS*=.FALSE. If *INPL* is .FALSE. *X* remains unchanged.

COMPLEX **Y(L*M*N)** [Output]

On output: if *INPL* is .FALSE. then *Y* contains the three-dimensional transformed data. If *LTRANS*=.TRUE. then the (ijk)th data element is stored in Y(i+(j-1)*L+(k-1)*L*M); otherwise, the (ijk)th data element is stored in Y(k+(j-1)*N+(k-1)*N*M). If *INPL* is .TRUE. then *Y* is not referenced.

COMPLEX **COMM(L*M*N+5*(L+M+N)+200)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence dimensions. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 3D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        SCALE = 1.0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL CFFT3DX(-1,SCALE,LTRANS,INPL,L,M,N,X,Y,COMM,INFO)
        IY = 1
        DO 20 I = 1, L
           DO 40 J = 1, M
              DO 10 K = 1, N
                 Y(IY) = Y(IY)*EXP(-0.001*REAL(I+J+K-3))
                 IY = IY + 1
   10         CONTINUE
   20      CONTINUE
           SCALE = 1.0/REAL(L*M*N)
           CALL CFFT3DX(1,SCALE,LTRANS,INPL,N,M,L,Y,X,COMM,INFO)
```

5.3 FFTs on real and Hermitian data sequences

The routines documented here compute discrete Fourier transforms (DFTs) of sequences of real numbers or of Hermitian sequences in either single or double precision arithmetic. The DFTs are computed using a highly-efficient FFT algorithm. Hermitian sequences are represented in a condensed form that is described in Section 5.1 [Introduction to FFTs], page 17. The DFT of a real sequence results in a Hermitian sequence; the DFT of a Hermitian sequence is a real sequence.

Please note that prior to Release 2.0 of ACML the routine ZDFFT/CSFFT and ZDFFTM/CSFFTM returned results that were scaled by a factor 0.5 compared with the currently returned results.

5.3.1 FFT of single sequences of real data

`DZFFT` Routine Documentation

DZFFT (*MODE,N,X,COMM,INFO*) [SUBROUTINE]

`INTEGER` **MODE** [Input]

The value of *MODE* on input determines the operation performed by `DZFFT`.
On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*= 1.

MODE=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to `DZFFT`.

MODE=2 : initializations and a real transform are performed.

`INTEGER` **N** [Input]

On input: N is the length of the real sequence X

`DOUBLE PRECISION` **X(N)** [Input/Output]

On input: X contains the real sequence of length N to be transformed.
On output: X contains the transformed Hermitian sequence.

`DOUBLE PRECISION` **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

`INTEGER` **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFT(0,N,X,COMM,INFO)
      CALL DZFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL ZDFFT(1,N,X,COMM,INFO)
```

SCFFT Routine Documentation

SCFFT (*MODE,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by SCFFT. On input:

MODE=0 : only initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*= 1.

MODE=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT.

MODE=2 : initializations and a real transform are performed.

INTEGER **N** [Input]

On input: *N* is the length of the real sequence *X*

REAL **X(N)** [Input/Output]

On input: *X* contains the real sequence of length *N* to be transformed.
On output: *X* contains the transformed Hermitian sequence.

REAL **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFT(0,N,X,COMM,INFO)
      CALL SCFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL CSFFT(1,N,X,COMM,INFO)
```

5.3.2 FFT of multiple sequences of real data

DZFFTM Routine Documentation

DZFFTM (*M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **M** [Input]
On input: M is the number of sequences to be transformed.

INTEGER **N** [Input]
On input: N is the length of the real sequences in X

DOUBLE PRECISION **X(N*M)** [Input/Output]
On input: X contains the M real sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed Hermitian sequences.

DOUBLE PRECISION **COMM(3*N+100)** [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFTM(1,N,X,COMM,INFO)
      CALL DZFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL ZDFFTM(1,N,X(1,3),COMM,INFO)
```

SCFFTM Routine Documentation

SCFFTM (*M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **M** [Input]
On input: M is the number of sequences to be transformed.

INTEGER **N** [Input]
On input: N is the length of the real sequences in X

REAL **X(N*M)** [Input/Output]
On input: X contains the M real sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed Hermitian sequences.

REAL **COMM(3*N+100)** [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFTM(1,N,X,COMM,INFO)
      CALL SCFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL CSFFTM(1,N,X(1,3),COMM,INFO)
```

5.3.3 FFT of single Hermitian sequences

ZDFFT Routine Documentation

ZDFFT (*MODE,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by ZDFFT. On input:

MODE=0 : only initializations (specific to the value of N) are performed; this is usually followed by calls to the same routine with *MODE*=1.

MODE=1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT.

MODE=2 : initializations and transform are performed.

INTEGER **N** [Input]

On input: N is length of the sequence in X

DOUBLE PRECISION **X(N)** [Input/Output]

On input: X contains the Hermitian sequence of length N to be transformed. On output: X contains the transformed real sequence.

DOUBLE PRECISION **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFT(0,N,X,COMM,INFO)
      CALL DZFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL ZDFFT(1,N,X,COMM,INFO)
```

CSFFT Routine Documentation

CSFFT (*MODE,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **MODE** [Input]

The value of *MODE* on input determines the operation performed by CSFFT. On input:

MODE=0 : only initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=1.

MODE=1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT.

MODE=2 : initializations and transform are performed.

INTEGER **N** [Input]

On input: *N* is the length of the sequence in *X*

REAL **X(N)** [Input/Output]

On input: *X* contains the Hermitian sequence of length *N* to be transformed. On output: *X* contains the transformed real sequence.

REAL **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFT(0,N,X,COMM,INFO)
      CALL SCFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL CSFFT(1,N,X,COMM,INFO)
```

5.3.4 FFT of multiple Hermitian sequences

ZDFFTM Routine Documentation

ZDFFTM (*M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **M** [Input]
On input: M is the number of sequences to be transformed.

INTEGER **N** [Input]
On input: N is the length of the sequences in X

DOUBLE PRECISION **X(N*M)** [Input/Output]
On input: X contains the M Hermitian sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed real sequences.

DOUBLE PRECISION **COMM(3*N+100)** [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFTM(1,N,X,COMM,INFO)
      CALL DZFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL ZDFFTM(1,N,X(1,3),COMM,INFO)
```

CSFFTM Routine Documentation

CSFFTM (*M,N,X,COMM,INFO*) [SUBROUTINE]

INTEGER **M** [Input]

On input: M is the number of sequences to be transformed.

INTEGER **N** [Input]

On input: N is the length of the sequences in X

REAL **X(N*M)** [Input/Output]

On input: X contains the M Hermitian sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed real sequences.

REAL **COMM(3*N+100)** [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER **INFO** [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFTM(1,N,X,COMM,INFO)
      CALL SCFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL CSFFTM(1,N,X(1,3),COMM,INFO)
```

6 ACML_MV: Fast Math and Fast Vector Math Library

6.1 Introduction to ACML_MV

ACML_MV is a library which contains fast and/or vectorized versions of some familar math library routines such as sin, cos and exp. The routines take advantage of the AMD64 architecture for performance, and so are currently only available with 64-bit versions of ACML. The routines in the library are very accurate over the range of acceptable input arguments.

Some of the performance is gained by sacrificing error handling or the acceptance of certain arguments. It is therefore the responsibility of the caller of these routines to ensure that their arguments are suitable. Furthermore, some of the routines are not callable from high-level languages at all, but must be called via assembly language; see the documentation of individual routines for details. Hence, these routines are intended to be utilized by knowledgeable users only.

6.1.1 Terminology

The individual documentation for a routine states what outputs will be returned for special arguments, and also gives an indication of performance of the routine. In general, special case arguments for any routine will cause a return value in accordance with the C99 language standard [6].

Special case arguments include NaNs and infinities, as defined by the IEEE arithmetic standard [7]. In these documents, *NaN* means *Not a Number*, *QNaN* means *Quiet NaN*, and *SNaN* means *Signalling NaN*.

A *denormal* number is a number which is very tiny (close to the machine arithmetic underflow threshold) and is stored to less precision than a normal number. Due to their special nature, operations on such numbers are often very slow. While such numbers might not necessarily be regarded as special case arguments, for the sake of performance some of the ACML_MV routines have been designed not to handle them. This has been noted in the documentation for each ACML_MV routine.

Performance of a routine is given in machine cycles, and is thus independent of processor speed.

Accuracy of a routine is quoted in *ulps*, where *ulp* stands for *Unit in the Last Place*. Since floating-point numbers on a computer are limited precision approximations of mathematical numbers, not all real numbers can be represented by machine numbers, and the machine number must in general be rounded to available precision. An *ulp* is the distance between the two machine numbers that bracket a real number.

In this document, the *ulp* is used as a measure of the error in a returned result when compared with the mathematically exact expected result. Because of the finite nature of machine arithmetic, a routine can never in general achieve accuracy of better than 0.5 *ulps*, and an accuracy of less than 1 *ulp* is good.

6.1.2 Weak Aliases

Some of the functions in ACML_MV include a weak alias to an equivalent function in libm. For example, the fastcos function includes a weak alias to cos. If ACML_MV is included in the link order before libm, then all calls to the aliased libm function name (e.g. cos) will use the equivalent ACML_MV routine (e.g. fastcos). If ACML_MV is included in the link order after libm, then all calls to libm functions will use the libm versions.

ACML_MV routines can always be accessed using their ACML_MV names (e.g. fastcos), regardless of link order.

6.1.3 Defined Types

The following types are used to describe the functions contained in this chapter:

__m128d a pair of double precision values;
__m128 four single precision values.

6.2 Fast Basic Math Functions

This section documents the interfaces to a set of basic mathematical functions.

fastcos: fast double precision Cosine

`double` **fastcos** (`double x`)

Weak alias: cos

C Prototype:

double fastcos (double x);

Inputs:

double x - the double precision input value.

Outputs:

Cosine of x.

Fortran Function Interface:

DOUBLE PRECISION FASTCOS(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

Cosine of X.

Notes:

fastcos computes the Cosine function of its argument x.

This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 2 ulp over most of the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

88 cycles for most valid inputs < 5e5.

fastsin: fast double precision Sine

`double` fastsin (`double x`)

Weak alias: sin

C Prototype:

double fastsin (double x);

Inputs:

double x - the double precision input value.

Outputs:

Sine of x.

Fortran Function Interface:

DOUBLE PRECISION FASTSIN(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

Sine of X.

Notes:

fastsin computes the Sine function of its argument x.

This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

88 cycles for most valid inputs < 5e5.

fastsincos: fast double precision Sine and Cosine

`void` **fastsincos** (`double x, double s, double c`)

Weak alias: sincos

C Prototype:

void fastsincos (double x, double s, double c);

Inputs:

double x - the double precision input value.

Outputs:

double s - Sine of x.

double c - Cosine of x.

Fortran Subroutine Interface:

SUBROUTINE FASTCOS(X,S,C)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Outputs:

DOUBLE PRECISION S - Sine of X.

DOUBLE PRECISION C - Cosine of X.

Notes:

fastsincos computes the Sine and Cosine functions of its argument x.

This function can provide a significant performance advantage for applications that require both the sine and cosine of an angle, such as axis and matrix rotation. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

99 cycles for most valid inputs < 5e5.

fastlog: fast double precision natural logarithm function

`double` fastlog (`double x`)

Weak alias: log

C Prototype:

double fastlog (double x);

Inputs:

double x - the double precision input value.

Outputs:

The natural logarithm (base e) of x.

Fortran Function Interface:

DOUBLE PRECISION FASTLOG(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

The natural logarithm (base e) of X.

Notes:

fastlog computes the double precision natural logarithm of its argument x.

This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

97 cycles for most valid inputs.
86 cycles for .97 < x < 1.03

fastlogf: fast single precision natural logarithm function

`float` fastlogf (`float x`)

Weak alias: logf

C Prototype:

float fastlogf (float x);

Inputs:

float x - the single precision input value.

Outputs:

The natural logarithm (base e) of x.

Fortran Function Interface:

REAL FASTLOGF(X)

Inputs:

REAL X - the single precision input value.

Return Value:

The natural logarithm (base e) of X.

Notes:

fastlogf computes the single precision natural logarithm of its argument x.

This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

94 cycles for most valid inputs.
85 cycles for .97 < x < 1.03

fastlog10: fast double precision base-10 logarithm function

`double` fastlog10 (`double x`)

Weak alias: log10

C Prototype:

double fastlog10 (double x);

Inputs:

double x - the double precision input value.

Outputs:

The base-10 logarithm of x.

Fortran Function Interface:

DOUBLE PRECISION FASTLOG10(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

The base-10 logarithm of X.

Notes:

fastlog10 computes the double precision base-10 logarithm of its argument x.

This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

112 cycles for most valid inputs.

fastpow: fast double precision power function

`double` **fastpow** (`double x, double y`)

Weak alias: pow

C Prototype:

double fastpow (double x, double y);

Inputs:

double x - the double precision base input value.

double y - the double precision exponent input value.

Outputs:

x raised to the power y.

Fortran Function Interface:

DOUBLE PRECISION FASTPOW(X,Y)

Inputs:

DOUBLE PRECISION X - the base value.

DOUBLE PRECISION Y - the exponent value.

Return Value:

X raised to the power Y.

Notes:

fastpow computes the x raised to the power y in double precision.

This is a *relaxed* version of pow, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs will produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0

$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	*NaN*
x<>1	*NaN*,	*NaN*

Performance:

200 cycles for most valid inputs.

fastpowf: fast single precision power function

`float` fastpowf (`float x, float y`)

Weak alias: powf

C Prototype:

float fastpowf (float x, float y);

Inputs:

float x - the single precision base input value.

float y - the single precision exponent input value.

Outputs:

x raised to the power y.

Fortran Function Interface:

REAL FASTPOWF(X,Y)

Inputs:

REAL X - the single precision base value.

REAL Y - the single precision exponent value.

Return Value:

X raised to the power Y.

Notes:

fastpowf computes the x raised to the power y in single precision.

This is a *relaxed* version of powf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs will produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0

$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	*NaN*
x<>1	*NaN*,	*NaN*

Performance:

175 cycles for most valid inputs.

fastexp: fast double precision exponential function

`double` fastexp (`double x`)

Weak alias: exp

C Prototype:

double fastexp (double x);

Inputs:

double x - the double precision input value.

Outputs:

e raised to the power x (exponential of x).

Fortran Function Interface:

DOUBLE PRECISION FASTEXP(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

e raised to the power X (exponential of X).

Notes:

fastexp computes the double precision exponential function of the input argument x.

This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

75 cycles for most valid inputs.

fastexpf: fast single precision exponential function

`float` fastexpf (`float x`)

Weak alias: expf

C Prototype:

float fastexpf (float x);

Inputs:

float x - the single precision input value.

Outputs:

e raised to the power x (exponential of x).

Fortran Function Interface:

REAL FASTEXPF(X)

Inputs:

REAL X - the single precision input value.

Return Value:

e raised to the power X (exponential of X).

Notes:

fastexpf computes the single precision exponential function of the input argument x.

This is a *relaxed* version of expf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

75 cycles for most valid inputs.

6.3 Fast Vector Math Functions

This section documents the interfaces to a set of vector mathematical functions.

vrd2_cos: Two-valued double precision Cosine

__m128d __vrd2_cos (__m128d x)

C Prototype:

__m128d __vrd2_cos(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

__m128d y - the double precision Cosine result pair, returned in xmm0.

Notes:

__vrd2_cos computes the Cosine function of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision Cosine of both values, returned as a __m128d value. This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

120 cycles for most valid inputs < 5e5 (60 cycles per value).

vrd4_cos: Four-valued double precision Cosine

`__m128d,__m128d __vrd4_cos (__m128d x1,__m128d x2)`

C Prototype:

__m128d __vrd2_cos(__m128d x);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

__m128d y1 - the first double precision Cosine result pair, returned in xmm0.

__m128d y2 - second double precision Cosine result pair, returned in xmm1.

Notes:

__vrd4_cos computes the Cosine function of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision Cosine of the four values, returned as two __m128d values. This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range. This routine may return slightly worse than 1 *ulp* for very large values between 4e5 and 5e5.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value).

vrda_cos: Array double precision Cosine

`void` **vrda_cos** (`int n, double *x, double *y`)

C Prototype:

void vrda_cos (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

Cosine for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_COS(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of Cosines of input values.

Notes:

vrda_cos computes the Cosine function for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes cos(x) for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value), n = 24.

vrd2_sin: Two-valued double precision Sine

`__m128d __vrd2_sin (__m128d x)`

C Prototype:

__m128d __vrd2_sin(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

__m128d y - the double precision Sine result pair, returned in xmm0.

Notes:

__vrd2_sin computes the Sine function of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision Sine of both values, returned as a __m128d value. This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

120 cycles for most valid inputs < 5e5 (60 cycles per value).

vrd4_sin: Four-valued double precision Sine

`__m128d,__m128d __vrd4_sin (__m128d x1,__m128d x2)`

C Prototype:

__m128d __vrd2_sin(__m128d x);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

__m128d y1 - the first double precision Sine result pair, returned in xmm0.

__m128d y2 - second double precision Sine result pair, returned in xmm1.

Notes:

__vrd4_sin computes the Sine function of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision Sine of the four values, returned as two __m128d values. This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range. This routine may return slightly worse than 1 *ulp* for very large values between 4e5 and 5e5.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value).

vrda_sin: Array double precision Sine

`void` **vrda_sin** (`int n, double *x, double *y`)

C Prototype:

void vrda_sin (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

Sine for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_SIN(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of Sines of input values.

Notes:

vrda_sin computes the Sine function for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes sin(x) for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range. This routine may return slightly worse than 1 *ulp* for very large values between 4e5 and 5e5.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value), n = 24.

vrd2_sincos: Two-valued double precision Sine and Cosine

`void __vrd2_sincos (__m128d x, __m128d* S, __m128d* C)`

C Prototype:

void __vrd2_sincos(__m128d x, __m128d* S, __m128d* C));

Inputs:

__m128d x - the double precision input value pair.

Outputs:

(Sine of x and Cosine of x.)

__m128d *S - Pointer to the double precision Sine result pair.

__m128d *C - Pointer to the double precision Cosine result pair.

Notes:

__vrd2_sincos computes the Sine and Cosine functions of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision Sin and Cosine of both values, returned as a __m128d value. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or application

not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

154 cycles for most valid inputs < 5e5 (77 cycles per Sine and Cosine of a value).

vrda_sincos: Array double precision Sine and Cosine

`void` **vrda_sincos** (`int n, double *x, double *ys, double *yc`)

C Prototype:

void vrda_sincos (int n, double *x, double *ys, double *yc)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*ys	- pointer to the array of sin output values.
double	*yc	- pointer to the array of cos output values.

Outputs:

Sine for each x value, filled into the ys array.

Cosine for each x value, filled into the yc array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_SINCOS(N,X,YS,YC)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION YS(N) - array of Sines of input values.

DOUBLE PRECISION YC(N) - array of Cosines of input values.

Notes:

vrda_sincos computes the Sine and Cosine functions for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes sincos(x) for each input value, and stores the results in the arrays pointed to by the ys and yc pointer inputs. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

180 cycles for most valid inputs < 5e5 (43 cycles per Sin and Cos of a value), n = 24.

vrd2_log: Two-valued double precision natural logarithm

__m128d __vrd2_log (__m128d x)

C Prototype:

__m128d __vrd2_log(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

The natural (base e) logarithm of x.

__m128d y - the double precision natural logarithm result pair, returned in xmm0.

Notes:

__vrd2_log computes the natural logarithm for each of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision natural logarithm of both values, returned as a __m128d value. This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

130 cycles for most valid inputs (65 cycles per value).

vrd4_log: Four-valued double precision natural logarithm

`__m128d,__m128d __vrd4_log (__m128d x1, __m128d x2)`

Prototype:

__m128d,__m128d __vrd4_log(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

The natural (base e) logarithm of x.

__m128d y1 - the first double precision natural logarithm result pair, returned in xmm0.

__m128d y2 - the second double precision natural logarithm result pair, returned in xmm1.

Notes:

__vrd4_log computes the natural logarithm for each of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision natural logarithm of the four values, returned as two __m128d values. This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

196 cycles for most valid inputs (49 cycles per value).

vrda_log: Array double precision natural logarithm

`void` **vrda_log** (`int n, double *x, double *y`)

C Prototype:

void vrda_log (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

The natural (base e) logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_LOG(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of natural (base e) logarithms of input values.

Notes:

vrda_log computes the double precision natural logarithm for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the natural log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

51 cycles per value for valid inputs, n = 24.

vrs4_logf: Two-valued single precision natural logarithm

__m128 __vrs4_logf (__m128 x)

C Prototype:

__m128 __vrs4_logf(__m128 x);

Inputs:

__m128 x - the single precision input values.

Outputs:

The natural (base e) logarithm of x.

__m128 y - the single precision natural logarithm results, returned in xmm0.

Notes:

__vrs4_logf computes the natural logarithm for each of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision natural logarithm of all four values, returned as a __m128 value. This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

124 cycles for most valid inputs (31 cycles per value).

vrs8_logf: Eight-valued single precision natural logarithm

`__m128,__m128 __vrs8_logf (__m128 x1, __m128 x2)`

Prototype:

__m128,__m128 __vrs8_logf(__m128 x1, __m128 x2);

Note that this function uses a non-standard programming interface. The two __m128 inputs, which contain eight single precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128 is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128 x1 - the first single precision input value pair.

__m128 x2 - the second single precision input value pair.

Outputs:

The natural (base e) logarithm of x.

__m128 y1 - the first single precision natural logarithm result pair, returned in xmm0.

__m128 y2 - the second single precision natural logarithm result pair, returned in xmm1.

Notes:

__vrs8_logf computes the natural logarithm for each of eight input arguments.

This routine accepts eight single precision input values passed as two __m128 values. The result is the single precision natural logarithm of the eight values, returned as two __m128 values. This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

200 cycles for most valid inputs (25 cycles per value).

vrsa_logf: Array single precision natural logarithm

`void` **vrsa_logf** (`int n, float *x, float *y`)

C Prototype:

void vrsa_logf (int n, float *x, float *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

The natural (base e) logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_LOGF(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of natural (base e) logarithms of input values.

Notes:

vrsa_logf computes the single precision natural logarithm for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes the natural log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

26 cycles per value for valid inputs, n = 24.

vrd2_log10: Two-valued double precision base-10 logarithm

__m128d __vrd2_log10 (__m128d x)

C Prototype:

__m128d __vrd2_log10(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

The base-10 logarithm of x.

__m128d y - the double precision base-10 logarithm result pair, returned in xmm0.

Notes:

__vrd2_log10 computes the base-10 logarithm for each of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision base-10 logarithm of both values, returned as a __m128d value. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

142 cycles for most valid inputs (71 cycles per value), longer for input values very close to 1.0.

vrd4_log10: Four-valued double precision base-10 logarithm

`__m128d,__m128d __vrd4_log10 (__m128d x1, __m128d x2)`

Prototype:

__m128d,__m128d __vrd4_log10(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

The base-10 logarithm of x.

__m128d y1 - the first double precision base-10 logarithm result pair, returned in xmm0.

__m128d y2 - the second double precision base-10 logarithm result pair, returned in xmm1.

Notes:

__vrd4_log10 computes the base-10 logarithm for each of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision base-10 logarithm of the four values, returned as two __m128d values. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

205 cycles for most valid inputs (51 cycles per value), longer for input values very close to 1.0.

vrda_log10: Array double precision base-10 logarithm

`void` **vrda_log10** (`int n, double *x, double *y`)

C Prototype:

void vrda_log10 (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

The base-10 logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_LOG10(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of base-10 logarithms of input values.

Notes:

vrda_log10 computes the double precision base-10 logarithm for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the base-10 log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

53 cycles per value for valid inputs, n = 24, longer for input values very close to 1.0.

vrd2_exp: Two-valued double precision exponential function

__m128d __vrd2_exp (__m128d x)

C Prototype:

__m128d __vrd2_exp(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

e raised to the power x (exponential of x).

__m128d y - the double precision exponent result pair, returned in xmm0.

Notes:

__vrd2_exp computes the exponential function of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision exponent of both values, returned as a __m128d value. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

80 cycles for most valid inputs (40 cycles per value).

vrd4_exp: Four-valued double precision exponential function

`__m128d,__m128d __vrd4_exp (__m128d x1, __m128d x2)`

Prototype:

__m128d,__m128d __vrd4_exp(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

__m128d y1 - the first double precision exponent result pair, returned in xmm0.

__m128d y2 - the second double precision exponent result pair, returned in xmm1.

Notes:

__vrd4_exp computes the double precision exponential function of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision exponent of the four values, returned as two __m128d values. This is a *relaxed* version of exp, suitable for use with fast-math compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

132 cycles for most valid inputs (33 cycles per value).

vrda_exp: Array double precision exponential function

`void` **vrda_exp** (`int n, double *x, double *y`)

C Prototype:

void vrda_exp (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

e raised to the power x (exponential of x) for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_EXP(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of exponentials (e raised to the power x) of input values.

Notes:

vrda_exp computes the double precision exponential function for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the e^x for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

33 cycles per value for valid inputs, n = 24.

vrs4_expf: Four-valued single precision exponential function

__m128 __vrs4_expf (__m128 x)

C Prototype:

__m128 __vrs4_expf(__m128 x);

Inputs:

__m128 x - the four single precision input values.

Outputs:

e raised to the power x (exponential of x) for each input value x.
__m128 y - the four single precision exponent results, returned in xmm0.

Notes:

__vrs4_expf computes the double precision exponential function of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision exponent of the four values, returned as a __m128 value. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

91 cycles for most valid inputs (23 cycles per value).

vrs8_expf: Eight-valued single precision exponential function

__m128,__m128 __vrs8_expf (__m128 x1, __m128 x2)

Prototype:

__m128,__m128 __vrs8_expf(__m128 x1, __m128 x2);

Note that this function uses a non-standard programming interface. The two __m128 inputs, which contain eight single precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128 is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128 x1 - the first single precision vector of four input values.
__m128 x2 - the second single precision vector of four input values.

Outputs:

__m128 y1 - the first four single precision exponent results, returned in xmm0.
__m128 y2 - the second four single precision exponent results, returned in xmm1.

Notes:

__vrs8_expf computes the single precision exponential function of eight input arguments.

This routine accepts eight single precision input values passed as two __m128 values. The result is the single precision exponent of the eight values, returned as two __m128 values. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

155 cycles for most valid inputs (19 cycles per value).

vrsa_expf: Array single precision exponential function

`void` **vrsa_expf** (`int n, float *x, float *y`)

C Prototype:

void vrsa_expf (int n, float *x, float *y)

Inputs:

int	n	- the number of single precision values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

e raised to the power x (exponential of x) for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_EXPF(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of exponentials (e raised to the power x) of input values.

Notes:

vrsa_expf computes the single precision exponential function for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes the e^x for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

15 cycles per value for valid inputs, n = 24.

vrs4_powf: 4-value vector single precision power function

__m128 __vrs4_powf(__m128 x,__m128 y)

C Prototype:

__m128 __vrs4_powf(__m128 x,__m128 y);

Inputs:

__m128 x - the single precision input base values.
__m128 y - the single precision input exponent values.

Outputs:

__m128 z - the single precision results of each x raised to the y power, returned in xmm0.

Notes:

__vrs4_logf() computes the single precision x raised to the y power for four pairs of input arguments. This routine accepts four single precision input value pairs passed as __m128 values. The result is the x raised to the y power for all four input pairs, returned as a __m128 value.

This is a *relaxed* version of powf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	*NaN*
x<>1	*NaN*,	*NaN*

Performance:

400 cycles for most valid inputs (100 cycles per value).

7 References

[1] C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Maths. Soft., 5 (1979), pp. 308–323.

[2] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson, *An extended set of FORTRAN basic linear algebra subroutines*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.

[3] J.J. Dongarra, J. Du Croz, I.S. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[4] David S. Dodson, Roger G. Grimes, John G. Lewis, *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 17 (1991), pp. 253–263.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philidelphia, (1999).

[6] Programming languages - C - ISO/IEC 9899:1999

[7] IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)

Subject Index

Routine Index